Expressing Multi-Way Data-Flow Constraint Systems as a Commutative Monoid Makes Many of their Properties Obvious

Jaakko Järvi
Texas A&M University, TX, USA
jarvi@cse.tamu.edu

Magne Haveraaen
University of Bergen, Norway
http://www.ii.uib.no/~magne/

John Freeman
Texas A&M University, TX, USA
jfreeman@cse.tamu.edu

Mat Marcus
Canyonlands Software Design
mmarcus@emarcus.org

Abstract

Here multi-way data-flow constraints systems are viewed as *commutative monoids*. A multi-way data-flow constraints system consists of a collection of constraints, each constraint being represented as a set of directed graphs. The monoid's binary operation between two constraints is defined as the set of (non-disjoint) graph unions between all possible pairs of graphs, choosing one graph from each constraint, such that the result of the union satisfies the conditions of a valid solution. This clearly is a commutative, associative operation, with the constraint containing the null graph as the neutral element. For a given constraint system, the carrier of the corresponding monoid is the closure of all combinations of constraints from the system. This then unifies the entire multi-way data-flow constraint system with a single constraint. This generic view is not at all a drastic departure of the established descriptions of multi-way data-flow constraint systems. Defining this generic view explicitly, however, makes several properties of and algorithms operating on constraint systems obvious. For example, a constraint system can be solved by folding the monoid's binary operator over the system's constraints. An example implementation in Haskell is described.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

General Terms Algorithms, Theory

Keywords Generic programming, data-flow constraint systems, code reuse

1. Introduction

A generic view of a data structure can help programmers to understand its properties and behavior, and design and implement algorithms operating on it. This paper reports experiences of one such situation: when representing a *hierarchical data-flow constraint system* [14] as a *commutative monoid*, several properties of data-flow constraint systems become straightforward. For example, solving a constraint system is that of a fold of the commutative monoid's binary operation over the system's constraints.

The reasons for the existence of this paper are two-fold. First, to serve as further evidence of the value of generic programming—even if there may be no significant immediate reuse gains, investing in understanding how a data structure can be adapted to conform to well-known algebraic structures is beneficial. Second, to describe the "commutative monoid-view" of data-flow constraint systems. This connection has to our knowledge not been presented before. It is possible that this is because it is not all that surprising. Maybe the authors working on constraint systems did not bother to make the connection explicit—the elements of such a connection, such as the operation of adding a constraint to an existing constraint system and its impact to the system's solution, have been described [6, 7].

The connection is, however, worth documenting—it simplifies and clarifies many aspects of multi-view data-flow constraint systems, and has helped us to advance our declarative approach to user interface programming [5, 9, 10] that is based on constraint systems.

Section 2 gives an account of multi-way data-flow constraint systems, and describes a (well-)known solving algorithm for such systems. The commutative monoid-view of constraint systems is described in Section 3. Section 4 shows a somewhat simplistic implementation of the "multi-way data-flow constraint system monoid" in Haskell. Section 5 describes how multi-way data-flow constraint systems arise in user interface programming. Further, this section highlights some benefits of the commutative monoid view. Section 6 concludes the paper.

2. Background

A constraint system consists of *variables* and *constraints*. Constraints are defined as relations between subsets of those variables; *solving* the constraint system means finding a valuation for the system's variables so that all relations in the system are satisfied. This section describes a class of constraint systems—hierarchical multi-way data-flow constraint systems [14]—and how they can be solved. (The "hierarchical" aspect is discussed in Section 5.) In particular, we detail how a data-flow constraint system can be represented as a graph. This graph representation is the basis for defining a commutative composition operation between two constraints,

WGP'12, September 9, 2012, Copenhagen, Denmark.

and ultimately thus a basis for the commutative monoid-view of constraint systems.

2.1 Multi-way data-flow constraint systems

A multi-way data-flow constraint system S is a tuple $\langle V, C\rangle$, where V is a set of variables and C a set of constraints. Each constraint in C is a tuple $\langle R, r, M\rangle$, where $R \subseteq V$, r is an n-ary relation ($n = |R|$) among variables in R, and M is a set of *constraint satisfaction methods*, or just *methods*. If the values of the variables in R satisfy r, we say that the constraint is *satisfied*. Executing any method m in M *enforces* the constraint by computing values for some subset of R, using another disjoint subset of R as inputs, such that the relation r becomes satisfied. We refer to the input and output variables of a method m as $ins(m)$ and $outs(m)$, respectively. The code realizing a method is considered a "black box"—it is the programmer's responsibility to ensure that a constraint is satisfied after any of its methods is executed.

The relations in constraint systems can be arbitrary, but often they are equalities. For example, the equation $r = w/h$ could describe what should hold true of the width w, height h, and aspect ratio r of an image. One possible set of methods for this equation is $M_1 = \{w \leftarrow rh, h \leftarrow w/r, r \leftarrow w/h\}$. Another relation might connect the width and height to the size of the file necessary to represent the image, say, $s = g(wh)$. We assume that the function g realizes the size calculation, and that its details are not known. For this relation a possible (singleton) set of methods is $M_2 = \{s \leftarrow g(wh)\}$; that is, the constraint can only be satisfied in one direction, by computing s from w and h.

The constraint satisfaction problem for a constraint system $S = \langle V, C\rangle$ is to find a valuation of the variables in V such that each constraint in C is satisfied. Such a valuation is attained by enforcing each constraint in turn, without changing the valuations of already enforced constraints. Concretely, exactly one method from each constraint in C is executed in an order in which no method writes to a variable that has already been read from or written to by another method.

A solution to a multi-way data-flow constraint system is thus characterized by a partially ordered set of methods. Each valid execution order of methods is often called a *plan* [7].

The next section develops the view of a constraint system as a graph. We begin with a "global" view of a graph for the entire constraint system and then discuss how to extract the subgraphs that correspond to individual constraints and individual methods. These will be the building blocks for the monoid-view, developed in Section 3. Our graph representation is a little different from prior works [14] where multi-view data-flow constraint systems are represented as undirected graphs with auxiliary information that expresses how the undirected graph can be directed. In our representation, such auxiliary data is unnecessary.

We note that the following well-formedness condition for constraint systems, known as *method restriction* [11, p. 56], is often assumed:

Definition 1 (Method restriction)**.** The method restriction holds for a multi-way data-flow constraint system $S = \langle V, C\rangle$ if for all constraints $\langle R, r, M\rangle$ in C, for all methods m in M, $\{ins(m), outs(m)\}$ is a partition of R.

In other words, each method of a constraint must use every one of the constraint's variables either as an input or output (but not both). We rely on this assumption in our graph constructions. Method restriction also enables a polynomial time solving algorithm [13].

2.2 Constraint system as a graph

A well-formed multi-way data-flow constraint system S is in a one-to-one correspondence with an *oriented*, *bipartite* graph $G_S = \langle V + M, E\rangle$, with vertex sets V and M representing the variables and methods of the system, respectively, and E the directed edges that connect each method with its input and output variables. Where $v, u \in V$ and $m \in M$, the edge (v, m) indicates that the variable v is an input of the method m, and (m, u) that m outputs to the variable u. The graph is oriented, that is, $(a, b) \in E \implies (b, a) \notin E$, because for each method m, $ins(m)$ and $outs(m)$ are disjoint. We call this graph the *constraint system graph*. Figure 1 shows the constraint system graph of the example constraint system discussed in Section 2.1.

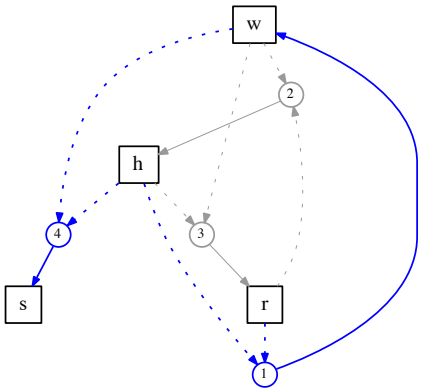

Figure 1. A constraint system graph of the two-constraint example constraint system discussed in Section 2.1. Variable vertices are shown as rectangles and method vertices as circles. Edges representing the inputs of a method are shown as dashed and outputs as solid lines. The variables h, w, and r together with the methods 1, 2, and 3 constitute the constraint M_1, and the variables h, w, and s with the method 4 constitutes the constraint M_2. The thicker lines show one possible plan for the constraint system: executing method 1 first, then method 4 will satisfy the system.

The grouping of methods and variables into constraints is not explicit in the representation $G_S = \langle V+M, E\rangle$—but it is uniquely determined by G_S alone without any auxiliary data. Consider how a constraint system graph is constructed. Let A be some constraint $\langle R_a, r_a, M_a\rangle$ of S. The subgraph of G_S that represents A is $G_S[R_a + M_a]$[1]. Consider further some method of A, say m_a. That, too, can be represented as a subgraph of G_S, the one containing the vertex for the method m_a, variable vertices for all of the method's input and output variables (R_a), and the edges connecting m_a with the vertices in R_a. This subgraph of G_S is $G_S[R_a + \{m_a\}]$.

We refer to a graph representing a method as a *method graph*, and to a graph representing a constraint as a *constraint graph*.

Remark 2.1. A constraint graph is a (non-disjoint) union of method graphs. A constraint system graph is a (non-disjoint) union of constraint graphs.

We qualify the union operation as non-disjoint above, as sometimes graph union is understood as an operation between two disjoint graphs that share no vertices or edges. To be precise, the graph union operation $\langle V_1, E_1\rangle \cup \langle V_2, E_2\rangle$ is defined in the canonical way as the union of vertices and edges: $\langle V_1 \cup V_2, E_1 \cup E_2\rangle$. We note that graph union is associative and commutative [2, §1.4].

[1] We use the notation $G[V]$ to indicate the *vertex-induced* subgraph of G. If V is a subset of G's vertex set, $G[V]$ is the graph whose vertex set is V and whose edge set includes all edges of G with both endpoints in V.

The sharing between two distinct method graphs or two distinct constraint graphs is limited to variable vertices. Two different method graphs of the same constraint system are always edge-disjoint, and so are two different constraint graphs of the same constraint system. Further, method vertices are also disjoint.

To recover the structure of the constraint system, the grouping of methods and variables into constraints, from a constraint system graph, we use the notion of a *neighborhood*[2] (nbh) of a vertex. Because of method restriction, all method vertices of the same constraint have the same neighborhood. We note that a constraint system cannot be solved if two of its constraints share the same set of variables, and thus we assume as a well-formedness condition that two methods from different constraints do not have the same neighborhood.

Formally, if $\sim_{nbh}$ is the equivalence kernel of nbh in M, defined by $m_1 \sim_{nbh} m_2 \Leftrightarrow nbh(m_1) = nbh(m_2)$, the method vertices of each constraint are an equivalence class in the quotient set $M/\sim_{nbh}$. Therefore, two methods m_1 and m_2 belong to the same constraint if and only if $[m_1]_{nbh} = [m_2]_{nbh}$.

In the following, we omit the subscript $\cdot_{nbh}$ and just write $[\cdot]$ and $\sim$. The elements of the quotient set $M/\sim$ thus correspond to the constraints of the original constraint system S. Hence, if m is a method vertex in G_S, $[m]$ is a set of method vertices in G_S, and it identifies a constraint in S.

2.3 Solution as a graph

To solve a multi-way data-flow constraint system one (1) computes the plan—which method from each constraint should be executed and in which order—and then (2) executes the methods according to the plan. The plan can be explicitly represented as a subgraph of the constraint system graph; we call it the *solution graph*. The notation *in-degree*$_G(v)$ means the number of in-edges of a vertex v in the graph G.

Definition 2 (Solution graph, valid graph)**.** Let $G = \langle V + M, E\rangle$ be a constraint system graph. Let $M' \subseteq M$. $G' = G[V + M']$ is a *solution graph* of G iff

(1) G' is *acyclic*;
(2) $\forall v \in V$.*in-degree*$_{G'}(v) \leq 1$; and
(3) $|M'| = |M/\sim|$.

If conditions (1) and (2) hold, we say that G' is a *candidate solution graph*.

The definition of a *candidate solution graph* is a convenience that allows us to refer to the first two conditions in a context where a full constraint system graph does not exist.

The second condition in the above definition is to guarantee that no two methods output to the same variable. There is no explicit condition disallowing the inclusion of more than one method from the same constraint—due to the method restriction, any such graph necessarily violates either the first or the second condition. The third condition states that there are as many methods as there are constraints, and thus a solution graph contains exactly one method from each constraint.

Executing each method in a topological order of the solution graph's method vertices will give the system's variables a valuation that satisfies all the system's constraints.

2.4 Computing the solution graph

In describing the algorithm for finding the solution graph we use the notion of a *free variable*, introduced in the original Sketchpad report [12], and extend the notion to methods as follows:

[2] If a is a vertex of a graph whose edges are E, then $nbh(a) = \{b \mid (a, b) \in E \text{ or } (b, a) \in E\}$.

Definition 3 (Free variable, free method)**.** Let $G = \langle V + M, E\rangle$ be a constraint system graph. A variable v is free in G if $\forall n, m \in M.n \in nbh(v) \wedge m \in nbh(v) \implies [n] = [m]$. A method m is free in G if $\forall v \in outs(m).v$ free.

Hence, a variable that belongs to exactly one constraint is free. A method that outputs to only free variables is free. A free method can enforce a constraint without restricting which methods can be used to enforce other constraints in the system.

A solution graph is computed by repeating the following steps: find a constraint with a free method, include the method in the resulting solution graph, and remove the constraint from the system. The process will either succeed to remove all constraints, in which case the solution graph has been identified, or fail if constraints remain but no free methods exist.

Algorithm 1 shows the pseudo-code. The state of the algorithm is captured in two sets of methods: M_s are the methods that are part of the solution graph/plan, and M_u the methods of the not yet satisfied constraints.

Algorithm 1 PLANNER$(G\langle V + M, E\rangle)$

```
1: M_s ← ∅, M_u ← M
2: while M_u ≠ ∅ do
3:     if no free methods in G[V + M_u] then
4:         return "no solution"
5:     m ← some free method in G[V + M_u],
       M_u ← M_u \ [m], M_s ← M_s ∪ {m}
6: return G[V + M_s]
```

Under-constrained systems will, during one or more rounds of the algorithm, have more than one free method in a single constraint to choose from, and thus the resulting solution graph of PLANNER may be ambiguous.

We note that "incremental" planner algorithms have been presented [7, 14]. An incremental planner algorithm computes a new plan for a constraint system after small modifications to the system, such as adding or removing constraints. Though faster in many cases, the worst-case time complexity of these algorithms is no better than that of the above simple planner.

3. Constraint system as a monoid

To view constraint systems as a commutative monoid, we consider methods and constraints to be graphs directly following the construction of method and constraint graphs as given in Section 2.2, summarized in Remark 2.1. That is, the method graph of a method $m \in M$ of some constraint $C = \langle R, r, M\rangle$ is a graph with one method vertex m, variable vertices R, and edges (r, m) for each $r \in ins(m)$ and (m, r) for each $r \in outs(m)$. To not confuse the different views of a method, an undecorated method name, say m, refers to the single method vertex, and a method name decorated with the superscript $\cdot^g$, say m^g, refers to the method graph.

Further, as indicated in Section 2.2, the graph of a constraint is the graph union of the graphs of the constraint's methods. Similar to methods, we decorate a constraint in the same way as a method to make clear our reference to a constraint's graph representation. The constraint graph of the above constraint C is thus $C^g = \bigcup_{m \in M} m^g$.

A single constraint, such as C above, is in itself a constraint system. Hence, the constraint graph of the constraint C and the constraint system graph of the constraint system consisting of the sole constraint C coincide.

We can also unify method graphs and solution graphs in such a single-constraint system. C can be satisfied by executing any of its methods; any of C^g's method graphs m^g will, by construction,

satisfy the requirements of Definition 2 and be a solution graph. This set of solution graphs, each a DAG with at most one in-edge to each variable vertex, represents all possible solutions of the constraint system arising from the single constraint C.

All of the above definitions apply also to constraint systems that arise from collections of several constraints. Consider a constraint system of two constraints, say, $A = \langle R_a, r_a, M_a \rangle$ and $B = \langle R_b, r_b, M_b \rangle$. If these two constraints are composed to a single "opaque" constraint, the variable set of this constraint is $R_a \cup R_b$ and the relation $r_a \wedge r_b$. The methods are all the (functional) compositions of one method from each of M_a and M_b, such that no variable is assigned a value twice, or assigned after being read.

The constraint system graph for the constraint system consisting of constraints A and B is simply $A^g \cup B^g$, and can be viewed as a single constraint. Each method of this constraint is $m_a^g \cup m_b^g$, for some $m_a \in M_a, m_b \in M_b$.

The set of constraint graphs as a carrier set, the null graph as the initial element (graph union with a null graph is an identity operation), and the graph union operation as the binary operation could be considered a monoid. This would, however, be a rather uninteresting construction—and it is not our objective. Merely composing two constraint graphs together to form a larger constraint graph reveals no information about the composed system. Whether this composed constraint (system) has any solutions or not is unknown without examining the constraint graph further, e.g., by solving the system with the PLANNER algorithm described in Section 2.4. Furthermore, above we said that each method of a composed constraint graph is a graph union between two methods, one from each of the component constraints. The resulting method may not be satisfiable; it can be cyclic or contain two in-edges for the same variable.

Our examination of the structure of multi-way data-flow constraint systems arose from the need to understand more of the behavior of a given constraint system than just finding a single solution. We apply constraint systems to user interface programming [5, 9, 10]; to understand all possible ways a user interface can react to user interaction relies on understanding the characteristics of the constraint system that determines the behavior of the user interface. A typical task includes examining what are all the possible solutions to a constraint system where only some constraints need to be enforced and others not, and what is the impact of adding a new constraint to the system. For these kinds of queries, a suitable representation is to retain the graphs of each method separate.

3.1 Definition of the constraint monoid

A constraint is a *set of method graphs*, each of which share the same set of variable vertices and satisfies the requirements of a candidate solution graph (Definition 2: a DAG with at most one in-edge for each variable vertex). This representation of constraints is the carrier of the commutative monoid we are defining.

Operationally, the composition operator between two constraints is then forming the cartesian product of the two sets of method graphs, computing the graph union of each pair of this product, and discarding those graphs that are not valid candidate solution graphs. Formally, let $A = \{a_1^g, a_2^g, \ldots, a_m^g\}$ and $B = \{b_1^g, b_2^g, \ldots, b_n^g\}$ be constraints (sets of method graphs). Then our monoid operation is defined as $A + B = \{c^g \mid a^g \in A, b^g \in B, c^g = a^g \cup b^g, c^g \text{ a candidate solution graph}\}$. This is both associative and commutative because of the associativity and commutativity of graph union.

The identity element is the singleton set, whose sole element is the null graph. This is easily seen:

$$\begin{aligned} & \{a_1^g, a_2^g, \ldots, a_m^g\} + \{(\{\}, \{\})\} \\ = \; & \{a_1^g \cup (\{\}, \{\}), a_2^g \cup (\{\}, \{\}), \ldots, a_m^g \cup (\{\}, \{\})\} \\ = \; & \{a_1^g, a_2^g, \ldots, a_m^g\}. \end{aligned}$$

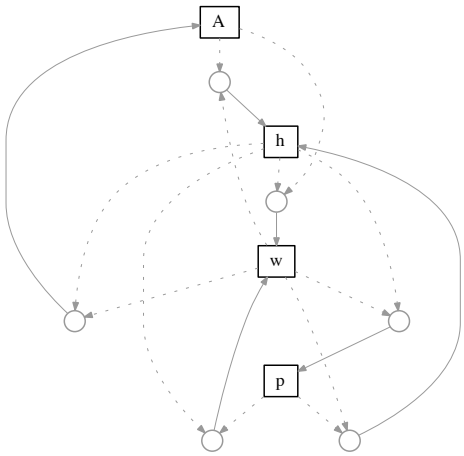

Figure 2. A constraint graph of a constraint composed of two other constraints, the first of which represents a three-way relation between A, w, and h; and the second also a three-way relation between p, w, and h.

Note that the empty set is an *absorbing element* of the constraint composition operator; the cartesian product of a set with an empty set is an empty set. Further, the $+$ operation is *idempotent*: for all constraints C, $C + C = C$. This follows, as the union of two different method graphs from the same constraint is cyclic, and thus the only method graphs that remain in the result are the unions of each method graph with itself; and graph union is idempotent.

The size of $A + B$ can be up to $m \times n$ methods. This is, for example, the case in the example from Section 2.1, illustrated in Figure 1. The individual constraints M_1 and M_2 have, respectively, three and one methods; The combined constraint $M_1 + M_2$ has 3×1 method graphs.

For it to be interesting to compose constraints, the constraints must share variables, often in ways that will disqualify many combinations of methods. Figure 2 shows the constraint graph arising from the composition of two constraints, each containing three variables; two of the variables are shared. The example from which these two constraints arise is modeling the relation of a rectangle's width w, height h, and area A; and, respectively, its width, height and perimeter p. Each constraint has three methods: any one variable can be computed from the other two. Instead of 3×3 methods, the combined constraint has five methods. Figure 3 shows all these method graphs.

4. A minimalistic implementation of the constraint monoid

As we explained in Section 2.2, all structure of a constraint system can be recovered from the constraint graph. That is, one does not need to maintain auxiliary information of which method vertices belong to which constraints; or of which "elementary" constraints the system was composed. We can thus leverage a general purpose graph library, and directly represent methods, constraints, and constraint systems as graphs. Here, we are using Haskell's Data.Graph.Inductive library, based on Erwig's *Functional Graph Library* (FGL) [4].

Figure 4 shows a Haskell implementation of the monoid defined in Section 3.1. We include the necessary module import statements, on lines 1–6, to make the code complete and executable. Various

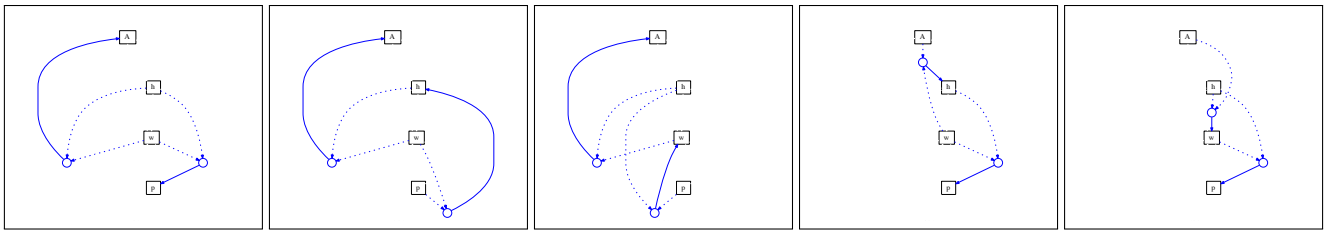

Figure 3. All methods graphs of the constraint graph in Figure 2.

modules are imported as *qualified*; this is to make it clear in which libraries various functions that we use are defined.

The graph, method, and constraint type definitions are shown on lines 8–12. The graph type that we use is G.Gr NodeKind (). The first type parameter of G.Gr represents the type of the label of nodes, the second the type of the label of edges. A node in the FGL graph is a pair of an integer and a label. The integer is a node's unique identifier. The label type NodeKind indicates whether a node represents a variable or a method. An edge is a triple, whose elements are an integer (source), integer (target), and a label. The source and target integers each should refer to an existing node identifier. We can do without any additional information on edges; hence the label type is unit.

The method and constraint types directly follow the definitions in Section 3.1: a method is a graph and a constraint a list of methods, also graphs. The general graph type Gr.G can represent any graphs, including ones that are not valid method graphs. Further, a list of method graphs might not constitute a valid constraint. We assume as an unchecked precondition that methods and and constraints are always well-formed. Indeed, constraints are typically constructed with dedicated functions (e.g., we construct constraint systems from specifications written in a domain-specific language [9]) that guarantee well-formedness.

The workhorse of the monoid's binary operation is the function methodUnion. This function computes the union of two graphs, but only accepts the result if it is a candidate solution graph. The FGL library does not provide a graph union function, so we implement here a simple graph union operation (methodUnion). The methodUnion function extracts both a list of nodes and a list of edges from its argument graphs, uses the list union function to combine the nodes and combine the edges, and reconstructs a graph from the so created list of nodes and list of edges.

The resulting graph is checked against Defintion 2 on lines 25 and 26. The first line is for checking that no variable is an output of more than one method. This check is only necessary for the variables that are common to both of the composed graphs. The function fst accesses the node index of a labeled node, G.indeg computes the number of incoming edges of a node. The second line is for checking whether the resulting graph contains cycles.

Taking advantage of the methodUnion function, Constraint can be made an instance of the Monoid type class without much effort. The identity element (called mempty in Haskell) is a singleton list of methods, where the method is the empty graph. The binary operation, mappend, applies methodUnion to each combination of the operands' methods, and collects the results in a list. The catMaybes function prunes the Nothing values that result from the failed method unions where the result does not satisfy the conditions of a solution graph.

With constraints defined as monoids, writing a planner algorithm becomes quite straight-forward. Assuming a constraint system is a list of constraints, the list of all possible plans/solution graphs for it, as a single Constraint, is obtained with the following function:

```
1  import Data.Monoid
2  import qualified Data.Graph.Inductive as G
3  import qualified Data.Graph.Analysis.Algorithms.Common as GA
4
5  import qualified Data.List as L
6  import Data.Maybe (catMaybes)
7
8  data NodeKind = VarNode | MetNode deriving (Eq, Show)
9
10 type Method = G.Gr NodeKind ()
11
12 data Constraint = Constraint [Method] deriving Show
13
14 methodUnion :: Method -> Method -> Maybe Method
15 methodUnion g1 g2 =
16   let ns1 = G.labNodes g1
17       ns2 = G.labNodes g2
18       es1 = G.labEdges g1
19       es2 = G.labEdges g2
20
21       common = L.intersect ns1 ns2
22
23       g = G.mkGraph (ns1 `L.union` ns2) (es1 `L.union` es2)
24   in
25     if all (<= 1) (map (G.indeg g . fst) common) &&
26        null (GA.cyclesIn g)
27     then Just g
28     else Nothing
29
30 instance Monoid Constraint where
31   mempty = Constraint [G.empty]
32   mappend (Constraint as) (Constraint bs) =
33     Constraint $ catMaybes [a `methodUnion` b | a <- as, b <- bs]
```

Figure 4. A minimalistic implementation of a data-flow constraint system planner in Haskell.

```
plans :: [Constraint] -> Constraint
plans = mconcat
```

Expanding mconcat out, plans = foldr mappend mempty.

5. Constraint Systems from User Interfaces: Constraint hierarchies

In our work on declarative user interface programming, we use constraint systems to model the pieces of data that are directly manipulated by a user interface. The role of the constraint system in our approach is that of the *ViewModel* in the *Model-View-ViewModel* pattern [8].

The idea is that distinct *views*, such as user interface widgets, are bound to variables in the constraint system. Modifying a widget causes a change in a variable of the constraint system, and thus invalidates one or more constraints. The constraint system is then solved, and the new values computed for the constraint system's variables are reflected in the views. In this manner, all the decisions on how to react to user input are delegated centrally to a constraint system, instead of following the contemporary practice of sprinkling such decisions among the widgets' event handling functions.

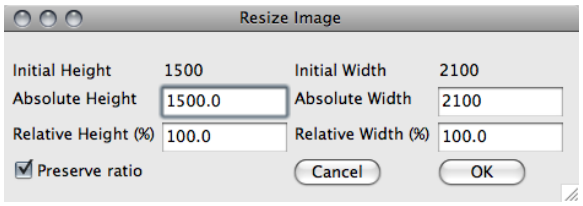

Figure 5. A simple user interface for scaling an image.

We describe the programming model, its reuse, and other benefits that ensue elsewhere [5, 9, 10]. Here, the interest is in the kinds of constraint systems that arise from this programming approach.

Figure 5 shows an example of a simple user interface with a handful of constraints. This user interface might be part of an image manipulation application, allowing one to scale an image. The avenues the user interface offers for this task are either to specify the image's absolute dimensions or dimensions relative to the initial dimensions, both of these with the option of whether to retain the original aspect ratio of the image.

A constraint system that can model the data manipulated by this user interface has one variable for each widget: *ih*, *ah*, and *rh* are for the initial, absolute, and relative height fields; *iw*, *aw*, and *rw* for the corresponding width fields; and *r* for the aspect ratio. The system has three constraints; Figure 6 shows the constraint system graph. There are eight possible solution graphs for this system, each corresponding to a different direction of propagating data among the widgets in the user interface.

The constraint system is under-constrained. A mechanism to determine the best solution graph, one producing the least-surprising behavior for the user interface, among multiple possible solution graphs is needed. A common approach is to introduce new constraints to make the constraint system over-constrained (a system for which no solution graph exists), and prioritize the satisfaction of some constraints over others. This technique is known as *constraint hierarchies* [3, 7, 14].

Constraint hierarchies assign a *strength* to each constraint. A plan of an over-constrained system is defined as the plan of the "best" system obtained from the over-constrained system by retracting some constraints. Intuitively, the best system is the one that retracts the weakest and fewest of the over-constrained system's constraints. Concretely, a *strength assignment* function $s : M/\sim \to \mathbb{N}$, maps a constraint to its strength. (Instead of $\mathbb{N}$, any totally ordered set could be used.) For two constraints c_1 and c_2, if $s(c_1) \geq s(c_2)$ then c_1 is at least as strong as c_2. We overload s to work for sequences of constraints as well: $s(c_1, \ldots, c_n) = s(c_1), \ldots, s(c_n)$. Then, of two sequences of constraints $\overline{C} = c_1, \ldots, c_r$ and $\overline{D} = d_1, \ldots, d_n$, both ordered in decreasing strength, $\overline{C}$ is stronger than $\overline{D}$ iff $s(\overline{C})$ is lexicographically greater than $s(\overline{D})$. This criterion is known as "locally-predicate-better" [7].

A well-known way to artificially make an under-constrained system become over-constrained is by introducing a *stay constraint* [14] for each variable in the system. A stay constraint consists of one variable and one method; the variable is an output of the method. Enforcing a stay constraint has no effect on the current valuation; a stay constraint keeps the value of its variable unchanged. Figure 7 shows the constraint graph of Figure 6 with the stay constraints added.

Unless the system has no other constraints, not all stay constraints can be satisfied. To select which constraints should be satisfied, and to avoid too many surprises to the user, stay constraints are assigned (unique) strengths based on how recently they have been edited. We can consider the constraints of the original under-

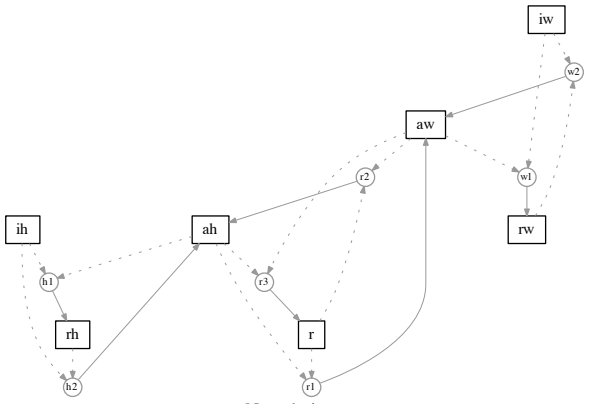

Figure 6. The graph of the constraint system for the user interface in Figure 5.

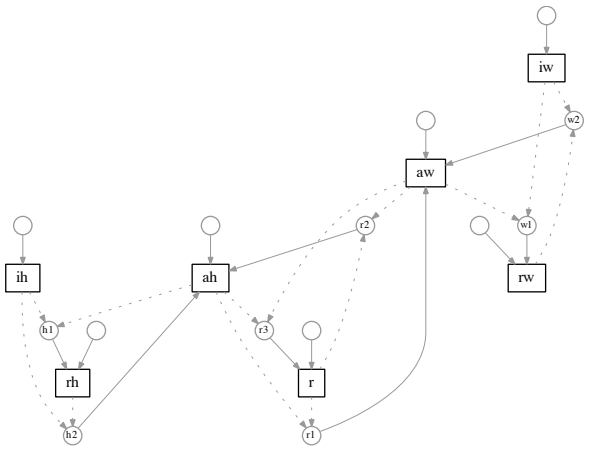

Figure 7. Stay constraints.

constrained system with each different subset of the stay constraints as a constraint system of its own. The strength assignment induces a total order of preference among these constraint systems, and in this way also a total order of preference among the solution graphs of the original under-constrained system. Figure 8 shows the best solution graph for the system of Figure 7 for the cases where *rh* and *r* have greatest strengths. The stay constraints of *ih* and *iw* are always enforced in every best solution.

The monoid-view makes it easy to study the strength assignment's impact to the solution. To find the best locally-predicate-better solution, one starts from the original under-constrained system as the current solution, and adds stay constraints to it in the order from the strongest to the weakest. The results of additions that fail (return the absorber element) are ignored. For concreteness, we express this in Haskell. The order of the list of constraints passed to prisolve determines the strength assignment:

```
isAbsorber (Constraint []) = True
isAbsorber _ = False

prisolve :: [Constraint] -> Constraint
prisolve ls = foldl1 (\a b -> let sum = a `mappend` b in
                              if isAbsorber sum then a else sum) ls
```

Understanding the mapping from strength assignments to the set of possible plans is quite desirable in user interface design.

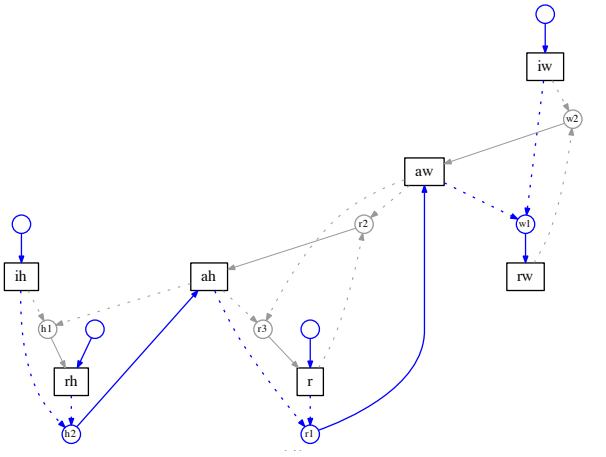

Figure 8. A constraint graph that satisfies stay constraints for *rh*, *r*, *ih* and *iw*.

Questions, such as "can the user interface ever override the value of any field that a user is editing?" or "if fields 1, 2, and 3 are edited in this order, will the user interface modify either 1 or 2; and if so, which?" are sometimes hard to answer conclusively based on complex event handling logic.

With the help of the monoid-view, the first question is answered by confirming that adding any of the stay constraints alone to the original under-constrained system yields a satisfiable constraint. Assuming the stays are the stay constraints of all variables, cs the original under-constrained constraint system (composed into one constraint), the query can be written in Haskell as any (isAbsorber . (mappend cs)) stays. The second question can be answered by adding the stay constraints of the fields 3 and 2 to the original system, then those of 3 and 1, and observing the possible solutions.

The impact of a change in the strength assignment is a source of interesting questions. In particular, what changes may require recomputing a new solution graph? Some properties that are obvious with the monoid view are:

- Increasing the strength of an enforced constraint does not change the best solution graph.
- Decreasing the strength of an unenforced constraint does not change the best solution graph.

Since the best solution graph is the sum of a sequence of constraints, which is commutative and associative, adding the same constraints in a different order yields the same solution. The question then is could strengthening an enforced constraint so that it surpasses some unenforced constraint in the preference order enable that constraint to be enforced. Let $[]$ denote the absorber constraint, and a, b, and c some constraints s.t. $s(a) \geq s(b) \geq s(c)$. If a and c are enforced in the best solution but b is not, then $a+b = []$ and $a + c \neq []$. Clearly $(a + b) + c = [] = (a + c) + b$ and thus b cannot be enforced even if c is added to the solution first. The same reasoning applies to the second case, weakening an unenforced constraint, here moving b later than c. We note that in the two related cases, decreasing (respectively increasing) the strength of an enforced (unenforced) constraint, the best solution graph may change.

In constraint graphs for user interfaces, increasing the strength of an enforced constraint is a common occurrence. Every time a user changes focus and updates a new variable, this variable is given the highest strength. A determination must be made whether the change can impact the best solution graph, that is, the direction of data propagation in the user interface.

Finally, the monoid view allows for specializing constraint systems. Even though there may be many constraints, the number of possible solution graphs of the entire constraint system may be small. For example, adding all three constraints of the system in Figure 6 into a single constraint yields eight methods. In cases where the number of solutions is not small for the entire system, there may be subsets of constraints for which there are only a small number of solutions. Such subsets of constraints are candidates for composing into a single constraint. Both the determination of which constraints should be composed and the actual composition are easy in the monoid view. Determining the optimal subsets to compose, however, can be computationally expensive with the rather simplistic constraint representation here.

In some cases it might be beneficial to get rid of the entire constraint solver. Consider HotDrink [1], our JavaScript library for programming user interfaces, which uses constraint systems as described above. When loading a page, the browser loads an implementation of a constraint system solver and a specification of a constraint system, both in Javascript. Typically the constraint system stays unchanged after loading, in which case an alternative is to translate the entire constraint system into a specialized sequence of branching instructions. When the total number of plans for a constraint system is small, the branching logic may be quite simple. As an example, of the eight possible solutions of the constraint graph in Figure 6, a stay constraint in the variable r and in any other of the four editable fields is enough to determine the solution. Without a stay constraint in r, a stay in either of the editable height fields and in either of the editable width fields suffices. The monoid-view is helpful for implementing these kinds of analyses and translations.

6. Discussion and Conclusions

Data-flow constraint systems are a well-trodden research area. Hierarchical constraint systems, the kinds of systems discussed in this paper, and solver algorithms for them were first presented decades ago. We started to learn about these systems when adopting them as the core of a model for declarative user interface programming. The descriptions of the systems found in the literature, several cited in this paper, are generally clear and well-written, so the learning task was reasonable.

In the prior works, as we do here, constraint systems are viewed as graphs. In prior works, the canonical representation of a constraint graph, however, is an undirected graph. Each constraint is represented with a single node, not with many method nodes, and connected to the variables of the constraint via undirected edges. Methods are then auxiliary information, possible ways to direct the undirected edges of each constraint [14].

Here, we map constraints to "vanilla" graphs. Even though no auxiliary data is stored, the constraints from which a constraint graph was constructed can be recovered from the graph. The benefit is that standard graph algorithms and libraries implementing those algorithms can be directly used in building software that operates on data-flow constraint systems. The implementations of the operations of the constraint system monoid and a simple constraint system planner algorithm based on the monoid are, we hope, indicative demonstrations of this.

The terminology of describing constraint systems in prior works has been developed, we suspect, along the development of the domain. While somewhat established and reasonably intuitive, we believe that the presented view of data-flow constraint systems as a monoid brings further clarity. Many properties become directly visible with no need for further inspection. For example, that constraints are a commutative monoid tells us immediately that sets of constraints can be composed to new constraints in any combina-

tions and composed in any order with no impact on the final result. Inferring such guarantees with less "generic" views of constraint systems requires a bit more inspection.

A quite visible activity of generic programming—judging from the contents of, say, Haskell or C++ (*de facto*) standard libraries—has been to encode well-known mathematical structures and algorithms based on those structures as executable program codes. Especially Haskell libraries go quite far in this direction, offering the (advanced) programmer an ever-expanding collection of abstractions to map concrete data structures. Benefits of generic programming materialize when such mapping takes place.

This paper provides a mapping of concrete data structures representing constraint systems into the rather simple monoid abstraction. In addition to offering this result, directly usable for programming tasks in the constraint systems domain, this paper is also an experience report on generic programming. New understanding of a domain can be obtained, learning of existing discoveries can be made easier, and programming tasks can be simplified through a process of analyzing the properties of concrete data structures and representing them as instances of well-known algebraic structures.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0845861.

References

[1] Hotdrink, an open source JavaScript GUI framework, Accessed June 2012. URL https://github.com/HotDrink.

[2] J. A. Bondy and U. S. R. Murty. *Graph Theory*. Springer, 2008.

[3] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf. Constraint hierarchies. *SIGPLAN Not.*, 22(12):48–60, Dec. 1987. ISSN 0362-1340. URL http://doi.acm.org/10.1145/38807.38812.

[4] M. Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, Sept. 2001. ISSN 0956-7968. URL http://dx.doi.org/10.1017/S0956796801004075.

[5] J. Freeman, J. Järvi, W. Kim, M. Marcus, and S. Parent. Helping programmers help users. In *GPCE'11: Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pages 177–184, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0689-8. doi: 10.1145/2047862.2047892.

[6] B. N. Freeman-Benson. A module mechanism for constraints in smalltalk. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '89, pages 389–396, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. URL http://doi.acm.org/10.1145/74877.74918.

[7] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990.

[8] J. Gossman. Introduction to Model/View/ViewModel pattern for building WPF apps, October 2005. URL http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx.

[9] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE'08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98, New York, NY, USA, 2008. ISBN 978-1-60558-267-2. doi: 10.1145/1449913.1449927.

[10] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Algorithms for user interfaces. In *GPCE'09: Proceedings of the 8th international conference on Generative programming and component engineering*, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1621607.1621630.

[11] M. J. Sannella. *Constraint satisfaction and debugging for interactive user interfaces*. PhD thesis, University of Washington, Seattle, WA, USA, 1994.

[12] I. E. Sutherland. Sketchpad: a man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, spring joint computer conference*, AFIPS '63 (Spring), pages 329–346, New York, NY, USA, 1963. ACM. URL http://doi.acm.org/10.1145/1461551.1461591.

[13] G. Trombettoni and B. Neveu. Computational complexity of multiway, dataflow constraint problems. In *IJCAI: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 358–365, Aug. 1997.

[14] B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 18(1):30–72, Jan. 1996. ISSN 0164-0925. URL http://doi.acm.org/10.1145/225540.225543.